For kmalloc, I basically reused all of the umalloc.c source code. Since I couldn't use sbrk() to request more memory for the kernel, I replaced that call with kalloc(). This means that when the kernel asks for more memory, kmalloc either gets already allocated memory from a free list or requests a whole page of memory from kalloc(). In order not to waste all of this page, the code breaks it down into sizeof(Header) chunks. So when the kernel calls kmalloc, it gets (n/sizeof(Header) + 1) chunks allocated to it. This is kept in a list of in-use chunks. When kfree is called, the chunks associated with that memory address are placed in the free list.

To implement mmap, I added another element to the process struct called maps which is just a linkedlist that contains all of the mappings for a specific process. I use kmalloc to allocate space for the linkedlist. A single node of the linkedlist contains elements that help keep track of the metadata such as length of mapping, mapping address, flags, fd, etc. Since we are only doing anonymous mappings without caring about the flags (getting implemented in project 5), the only two arguments I need to take into account are address and length. So when a process calls mmap, the first thing that I do is ensure that the inputs are valid: ensure len > 0. If it is > 0, then I round the length up to the nearest page. I do this since all memory blocks returned by mmap need to be page aligned. I then check the address hint. If the address hint is >= KERNBASE and > p->sz, then the address I will try to map it to is PGROUNDUP(addr_hint). If the hint does not satisfy those restrictions, then I by default will try to map it to an address base I created specifically for mmap called MMAPBASE (0x40000000). Once I have a target_address mapping, I use walkpgdir to check if the mappings from target_address to target_address + length already exist (check if it exists in the process pgdir and if the PTE_P flag is set). If any of them already exist, then it increases the target_address by length and repeats the previous step. It does this once it either finds a free chunk of memory that has not been mapped or it reaches KERNBASE. If it reaches KERNBASE then it returns an error. If it finds a free chunk of memory that can be mapped, then it allocates memory and maps it to target_address using allocuvm(). If this is successful, then it creates a new linkedlist node to store all of the metadata for this mapping and adds it to the process' mmap linked list, p->mmaps. Munmap is a bit more straightforward. All I do is iterate through the process' mmaps linked list to check if the address that is passed is a valid mapping. If it finds that address in the linked list, then it calls deallocuvm() to free that memory and unmap it from the process' address space. Once that is done, it then removes the mapping's node from the mmaps linkedlist and frees that node using kmfree(). One list thing I added is code in allocproc() to initialize the process' mmaps linked list to empty (null) and code in freevm() to destroy the process' mmaps linkedlist by calling kmfree() for all the nodes in the list. There is no need for me to free and unmap the mmap mappings since that is already done in freevm().